Virtual Cluster Scheduling Through the Scheduling Graph

Josep M. Codina[1,2], Jesús Sánchez[2], Antonio González[1,2]
1. Dep. of Computer Architecture, UPC, Barcelona, Spain
2. Intel Barcelona Research Center, Intel Labs, UPC, Barcelona, Spain
E-mail: josep.m.codina@intel.com; f.jesus.sanchez@intel.com; antonio.gonzalez@intel.com

Abstract

This paper presents an instruction scheduling and cluster assignment approach for clustered processors. The proposed technique makes use of a novel representation named the scheduling graph which describes all possible schedules. A powerful deduction process is applied to this graph, reducing at each step the set of possible schedules. In contrast to traditional list scheduling techniques, the proposed scheme tries to establish relations among instructions rather than assigning each instruction to a particular cycle. The main advantage is that wrong or poor schedules can be anticipated and discarded earlier.

In addition, cluster assignment of instructions is performed using another novel concept called virtual clusters, which define sets of instructions that must execute in the same cluster. These clusters are managed during the deduction process to identify incompatibilities among instructions. The mapping of virtual to physical clusters is postponed until the scheduling of the instructions has finalized. The advantages this novel approach features include: (1) accurate scheduling information when assigning, and, (2) accurate information of the cluster assignment constraints imposed by scheduling decisions.

We have implemented and evaluated the proposed scheme with superblocks extracted from SpecInt95 and MediaBench. The results show that this approach produces better schedules than the previous state-of-the-art. Speed-ups are up to 15%, with average speed-ups ranging from 2.5% (2-Clusters) to 9.5% (4-Clusters).

1. Introduction

Clustering is becoming a common trend in the design of current microprocessors due to its ability to alleviate power-, thermal- and complexity-related problems faced by today's computer architects. Clustering consists of partitioning processor resources into several groups or clusters. The components of each cluster are simpler, faster, and consume less power than a traditional unified implementation. The resources in a cluster can be laid out in close proximity, which reduces signal transmission delays [15]. Long (and slow) wires are used to interconnect clusters.

Clustering is especially popular in DSP designs, including Texas Instruments' TMS320C6x [28], Analog Devices' TigerSHARC [13], BOPS's ManArray [25], HP/ST's Lx [11] and Equator's MAP1000 [14]. All of these processors use a statically-scheduled, clustered microarchitecture.

The effectiveness of a statically-scheduled processor strongly depends on the effectiveness of the compiler. Among the different compiler steps, code scheduling is probably the most critical one for the performance of these processors. In this paper, we focus on instruction scheduling techniques for clustered microprocessors. In particular, we focus on scheduling and cluster assignment for superblocks [16].

One key task to be performed, while scheduling code for clustered processors, is cluster assignment. The performance of clustered processors strongly depends on the ability of the compiler to assign instructions to the appropriate cluster so that workload is balanced and the effect of inter-cluster communications is minimized.

In this work, an instruction scheduling scheme for clustered architectures is proposed. The primary objective of the proposed algorithm is to generate schedules with high instruction-level parallelism while minimizing the penalties of inter-cluster communications. The technique is evaluated using more than 60,000 superblocks taken from several SpecInt95 and MediaBench applications. Results presented show that the proposed algorithm outperforms a recently proposed state-of-the-art technique (CARS [18]).

A key feature of the proposed algorithm is delayed cluster assignment. Rather than separating scheduling and cluster assignment into different steps [10][3][17][9][6][20], or integrating them and making decisions on scheduling and assignment at the same time for each individual instruction [24][18][21][19], the technique proposed in this paper integrates both tasks into a single step but decisions on scheduling and assignment are made in separated stages.

First, scheduling decisions drive the code generation process. The *consequences* of each scheduling decision are captured through the deduction process (DP); including constraints imposed to the cluster (e.g., instructions I_1 and I_2 cannot be assigned to the same cluster as an assignment consequence of deciding to schedule them in the same cycle, on a 2-issue machine).

Consequences obtained through the DP, based on the scheduling decisions, generate a partial cluster

International Symposium on Code Generation and Optimization (CGO'07)
0-7695-2764-7/07 $20.00 © 2007 IEEE

assignment. Virtual clusters are used to manage this partial cluster assignment. The use of the DP to identify the consequences of each scheduling decision provides the proposed scheme with an accurate mechanism to understand the impact of these decisions on the clustering assignment.

After all scheduling decisions have been made, the final stage of the algorithm performs the actual mapping of virtual clusters into physical ones. Hence, the cluster assignment is postponed until the whole schedule is available for making clustering decisions.

The DP is the core of the novel algorithm described in this paper. At each scheduling and assignment stage, one or more candidate decisions are heuristically selected. Selecting a decision induces consequences, further constraining the resulting schedule. These consequences are identified through the DP. Then, heuristics evaluate the resulting scheduling concluded by the DP, and the best decision is chosen or discarded.

Finally, in contrast to traditional list scheduling techniques, the scheduling decisions our proposal considers establish relations among instructions rather than assigning each instruction to a particular cycle (e.g., schedule instructions A and B in the same cycle without establishing the cycle). In order to manage such relations the Scheduling Graph, a new representation that allows for the enumeration of all possible schedules, is used.

The remainder of this paper is organized as follows. Section 2 provides a background on scheduling and clustering. Section 3 describes the scheduling graph, the virtual cluster graph which encodes the virtual clusters' relationships; and the deduction process. Next, the use of these tools is leveraged in Section 4 where the proposed technique is detailed. Then, Section 5 presents an example of the proposed technique, and Section 6 its experimental evaluation. Section 7 discusses related work and Section 8 concludes.

2. Background

2.1. Clustered Processors

In this work, a statically-scheduled clustered microarchitecture is considered. Each cluster is composed of multiple functional units and a register file. Clusters communicate register values among themselves using special copy instructions and a set of dedicated register buses. The memory hierarchy is centralized and shared by all clusters. In this work, we have assumed homogeneous clusters, although the proposed technique can be extended to deal with heterogeneous configurations. VLIW instructions flow through all clusters in a lockstep fashion (all clusters work on the same VLIW instruction together).

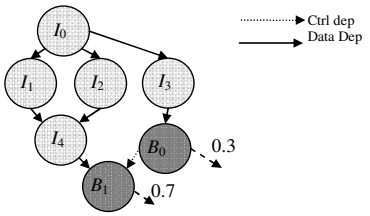


Figure 1. Example of a Superblock DG

2.2. Superblock Scheduling

A superblock consists of a sequence of consecutive basic blocks with a single entry point, and one or more exit points. Exit points from a superblock are branches and jump instructions. When scheduling superblocks the main objective is to minimize the average weighted completion time (AWCT). This metric represents the number of cycles between the entry point and each exit weighted by the exit probability, and it is computed as: $AWCT=\Sigma(Cyc_u+\lambda_u)\cdot P_u, \forall u \in Exit(DG)$ where Exit(DG) refers to the exit points from the superblock, P_u stands for the probability of the exit u to be taken, and Cyc_u is the cycle where instruction u has been scheduled. Finally, λ_u is the latency of instruction *u*.

In Figure 1 a superblock dependence graph (DG) is shown, composed of 3-cycle branch instructions (B_n) and 2-cycle non-branch instructions (I_n). Next to each branch, the probability of exiting the superblock is shown. Assuming that B_0 is scheduled in cycle 4 and B_1 in 6, then $AWCT=7\cdot0.3+9\cdot0.7=2.1+6.3=8.4$.

When all superblock exits are scheduled at their earliest start (*estart*), the lower bound for the execution of this superblock is achieved. This minimum AWCT (minAWCT) can be computed by considering the critical path and the resource constraints between the entry and all exit points, assuming no penalties due to inter-cluster communications.

Finally, the contribution in number of cycles of a superblock S to the execution of an application is computed by $TC(S) = AWCT(S)\cdot T(S)$ where T(S) is the number of times S is executed.

3. Novel Cluster Scheduling Mechanisms

In this section the main ingredients used in the proposed algorithm are described. The proposed technique: (1) makes use of scheduling graphs (SG), a representation that describes all possible schedules, and (2) postpones complete cluster assignment to the end of the scheduling process by using virtual clusters. Delaying the cluster assignment allows the algorithm to have more information about the schedule when making decisions on cluster assignment.

Scheduling graphs and virtual clusters are managed by a powerful deduction process (DP). This process captures an important amount of mandatory constraints induced by any decision made during the algorithm.

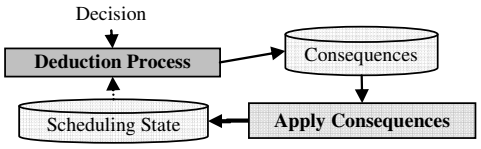


Figure 2. Deduction process usage

A decision refers to a selected action that further constraints the partial schedule and assignment. By taking several decisions a final schedule and assignment is obtained. In our approach, a decision may be one of the following actions: (1) establish a distance relation between two instructions in the final schedule (the final schedule is found when all instructions are assigned to a particular cycle and to a physical cluster), (2) schedule an instruction in a particular cycle, (3) assign a set of instructions to the same physical cluster, or (4) assign two sets of instructions to different physical clusters.

By using the DP to make decisions, the proposed scheme can select consequence-conscious decisions that improve the final schedule, while avoiding decisions that tend to invalid schedules.

The DP can be seen as a black box, as shown in Figure 2. Given a decision that is suitable to be made in the current state (see Section 4.3), the DP performs an analysis that produces a set of consequences.

A consequence refers to either a mandatory change over the current state or a contradiction. A mandatory change is a constraint that appears in any valid schedule build from the scheduling state created by the decision that triggers the mandatory change. A contradiction arises when the DP finds a situation where no valid schedule will be found no matter what decisions can be made in the future. E.g., an estart of an instruction is higher than its latest start (*lstart*).

The DP is not only responsible for discovering mandatory constraints, but it is also responsible for maintaining the coherence of the scheduling state when the consequences are applied. In particular, the DP is responsible for introducing the required communications when the producer and the consumer of a value are mapped into different physical clusters.

3.1. Scheduling Graph

A scheduling graph describes all possible schedules. To do that, the SG contains all feasible combinations between pairs of instructions that may overlap in any final schedule.

A combination is defined as a relation between a pair of instructions in terms of cycle-distance in the final schedule. Combinations exist between instructions that may overlap in any final schedule. Hence, the scheduling graph describes alternatives a scheduling algorithm may examine to find a valid schedule.

In Figure 3, all possible combinations between instructions B and I are shown. Assuming B is 3-cycle latency and I is 2-cycle latency, 4 combinations are possible. Given a unique identifier for each instruction and a lexicographic order among them, combinations between a pair of these instructions are named by the distance this combination induces in the final schedule. With such identifiers the combinations in the example of Figure 3 can be grouped into three categories:

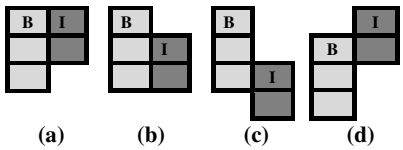


Figure 3. All combinations between B and I: (a) comb=0; (b) comb=-1; (c) comb=-2; (d) comb=1

- comb=0, if scheduled in the same cycle (Figure 3.a).
- comb<0, if the bigger instruction is scheduled |comb| cycles earlier. E.g., Figure 3.b and Figure 3.c.
- comb>0, if, following the lexicographical order, the smaller instruction is scheduled comb cycles earlier. E.g., assuming B< I, comb=1 is shown in Figure 3.d.

Note that only combinations in which an overlapping exists are of our interest. Hence, in the example of Figure 3, combinations with an id higher than 1, or lower than -2 are not considered a combination.

The process that performs scheduling through the SG consists of choosing some *feasible* combinations while discarding the others. In particular for each instruction pair either only one combination is selected or all of them are discarded. When a combination is chosen, a complex instruction is formed. This complex instruction is called a connected component (CC).

A combination is feasible if there is at least one valid final schedule where it appears. Given an instruction pair (u,v), a certain combination among them may appear in a final schedule depending on:

- **Dependences.** No feasible combination exists between instructions u and v when any direct or indirect relationship in the DG implies that u must be executed before v.
- **Resource conflicts.** A combination is not feasible if it implies any resource conflict.
- **AWCT.** Each combination imposes some constraints to the bounds of the instructions it involves. Combinations that meet resource and dependence constraints may be feasible for several AWCT values. However, a particular AWCT imposes some constraints on the bounds of these instructions which may not be compatible with the constraints in the bounds imposed by a particular combination.

The proposed scheme, as later described in Section 4, iteratively searches for a valid schedule setting at each step a targeted AWCT value. Since the AWCT varies at each step, only dependence and resource constraints, common for all AWCT values, are considered when

computing the SG. Hence, the number of combinations the SG must contain is limited and depends on: (1) the dependences in the dependence graph, (2) the resource constraints imposed by the tailored architecture; and (3) the instruction' latencies.

Given the DG = {V, E, λ}, we define the SG={V, E'} as an undirected graph such that an edge (u,v)∈E' exists if there is any feasible combination between u and v. Information about combinations is kept with the edge.

Figure 4.a shows the SG for the DG in Figure 1, assuming 3-cycle latency branches (B_i), 2-cycle latency non-branch instructions (I_j), and a 1-cluster architecture able to issue 2 non-branch and 1 branch instructions per cycle. Each edge in the SG is labeled with a number that indexes the table of Figure 4.b, where all feasible combinations in any AWCT are enumerated. In addition, next to each node, the (estart, lstart) bounds are shown. The lstart of each node is encoded by using the length between the entry of the superblock and the first exit that requires the execution of that instruction. In Figure 4.a, this length is referred to as LB_x where B_x is an exit instruction. This particular encoding allows for a single computation of the SG, and its reuse for all possible AWCT values.

In the example, feasible combinations for any AWCT exist: between B_0 and B_1, and between any instruction pair (u,v) where u does not depend on v (e.g., the pair (I_4, I_1) has no edge because I_4 is a successor of I_1). And, all possible combinations between two instructions connected through an edge in the SG, where at least one of them is of class I, are feasible. Resource constraints prevent combination 0 between a pair of branches, i.e., the machine allows a single branch per cycle.

3.2. Virtual Cluster Graph

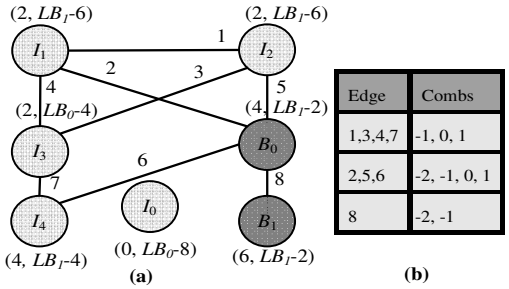
A virtual cluster (VC) refers to a set of nodes that will be mapped into the same *physical cluster* (PC). VCs are used to keep mandatory constraints imposed to the final cluster assignment by scheduling decisions.

In our approach, the process of mapping VCs into PCs is done after all scheduling decisions have been made. Hence, VCs are used to prevent and avoid scheduling decisions that may end up in non-valid or poor cluster assignments.

While decisions are made, it may happen that: (1) two VCs must be assigned to the same PC, or (2) two VCs cannot be assigned to the same PC. In the former case, the two VCs are *fused* into a new VC, whereas in the latter, both VCs are marked as *incompatible*.

A pair of VCs is incompatible if they *must* be mapped into a different PC. In order to keep the relationships among all VCs, the *virtual cluster graph* (VCG) is used.

The VCG is an undirected graph, where each node is a VC and edges link pairs of *incompatible* VCs. When



Edge	Combs
1,3,4,7	-1, 0, 1
2,5,6	-2, -1, 0, 1
8	-2, -1

Figure 4. Example of a Scheduling Graph

two VCs are incompatible, a value produced in one of them and consumed in the other requires a communication.

After making all scheduling decisions and during the postponed process that maps VCs into PCs, whenever the number of the VCs is lower than or equal to the number of PCs, a valid cluster assignment has been found. However, not all VCGs can be mapped into a given number of PCs, due to the constraints imposed by the edges in the VCG.

The existence of *cliques* of degree higher than the number of PCs creates a situation where no valid mapping between VCs and PCs can be found.

A *clique* is a fully connected subgraph. Hence, when a clique *C* is found in a VCG, the clusters belonging to *C* must be mapped to different clusters. Thus, if the number of VCs included in C is higher than the number PCs, no possible mapping of virtual clusters in VCG to physical ones can be found.

Although, analyzing all conditions that make a VCG unable to be mapped to a particular architecture is an NP-Complete problem, the most common situations where this occurs can be discovered and prevented. The technique proposed in this paper includes mechanisms in order to avoid the appearance of cliques of a degree higher than the number of PCs. In particular, the proposed technique features: heuristics to select the most appropriate decision (see Section 4.4.3), that takes into account the number of edges in the VCG to prevent this situation; and, a process to detect cliques based on a graph coloring scheme [4], applied after a decision has been studied by the deduction process and selected by the heuristics. If this process identifies a clique in the VCG, the decision is discarded.

In Figure 5 an example of the mapping process is shown. The VCG in Figure 5.a, obtained through the scheduling process, contains six VCs and nine edges describing incompatibilities among them. Assuming a target processor with four clusters, the mapping process works as follows: (1) Fuse VC_2 and VC_3, which results in VC'_2 (Figure 5.b); (2) Fuse VC_1 and VC_4, which results in VC'_1 (Figure 5.c); (3) Map VC_0 to PC_0, VC'_2 to PC_1, VC'_1 to PC_2, and VC_5 to PC_3 (Figure 5.d).

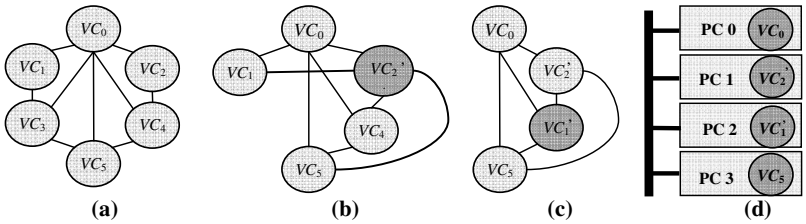


Figure 5. An example of a maping process from virtual to physical clusters

After fusing a pair of VCs, a new VC is created that contains all nodes of the original VCs, and an edge to any VC previously linked to the VCs fused. In the previous example, VC'_2 obtained from the fusion of VC_2 and VC_3 inherits all instructions previously mapped to them, and all edges from VCs linked to VC_2 or VC_3.

3.3. Deduction Process

The deduction process (DP) is the core of the proposed technique. Any decision that we may consider is submitted to the DP. The main objective of this process is to obtain the maximum amount of mandatory consequences of a decision.

The decision and its consequences induce a new partial scheduling and a new partial assignment. In our approach, some alternative new partial scheduling and assignment are generated with the DP, and then the best one is selected by the heuristics shown in Section 4.4.3.

The result of applying the DP may be a set of mandatory changes to the scheduling state (i.e., consequences), or a contradiction. A contradiction occurs when the DP finds: (a) an instruction with an estart higher than its lstart; (b) a combination that must be discarded for a given reason but chosen for another one; or, (c) a pair of VCs that must be fused for one reason but incompatible for another one.

The DP is an iterative process that treats a change, or a decision, on the current state at each step. Initially, the changes imposed by the decision submitted to the DP are considered. For each treated change, a set of rules is applied. A rule studies the change in the current state and concludes with some of its consequences, i.e., new changes on the state or a contradiction. Any change to the state concluded by a rule is further considered by the whole set of rules, so that, consequences of consequences may be obtained. The DP ends when no decision remains to be treated by the set of rules.

By defining the appropriate set of rules, a powerful DP can be built. Having a DP able to extract all consequences of any decision made allows for more informed decision making, which minimizes the negative impact of the use of heuristics However, defining a set of rules able to find all consequences for any decision is not feasible. Instead, our proposal includes several rules obtained by combining our knowledge of the cluster scheduling problem, and an intensive study of the characteristics of the SG and the VCG. Altogether the proposed rules result in a cost-effective solution.

The proposed set of rules can be divided in two main groups: state updating rules and deduction rules. The former rules are responsible for updating the scheduling state after any decision made, e.g., propagating any change on the scheduling bounds or including any required communication when a pair of VCs becomes incompatible. Deduction rules look for consequences that cannot be obtained with a simple update of the state according to a particular decision, but will be mandatory for any valid schedule that can be built based on applying the current decision to the scheduling state.

Among all rules, some treat scheduling while others deal with cluster assignment. The main objective of the set of deduction rules for scheduling is to deal with the problem of the interaction between dependences and resources. In particular, they look for resource usage requirements that may change the bounds of certain instructions, and select or discard some combinations. By using these resource-aware rules, the DP can anticipate resource conflicts and better understand the consequences of each considered decision for the final schedule. A more detailed description of the all rules can be found in [8]. The following section describes a representative selection of rules for cluster assignment.

3.3.1. Rules for Cluster Assignment

Cluster assignment rules look for incompatibilities and mandatory fusions between pairs of VCs. A pair of VCs must be fused when no slack or resources are available for the communication required by an outedge, whereas a pair of VCs becomes incompatible when no resources are available to accommodate them in the same cluster. Rules devoted to assignment detect these situations as they arise. Among them, two basic updating rules have been used, which snoop changes on the current state:

Rule 1: When an instruction's bound changes, if it has predecessors or successors belonging to a different VC and the change implies that there are not cycles enough to accommodate a communication between them, then: **Fuse both VCs**.

Rule 2: When a combination is chosen that links instructions from different VCs so that there are insufficient resources to allow the VCs to be fused, then: **VCs become incompatible**.

More advanced rules are used to study resource usages in pairs of VCs to found incompatibilities, and others deal with restrictions imposed by cluster assignment decisions to the scheduling. Examples of this latter class are the rules used to handle the assumption made that only one communication is allowed for each value (more communications may help register pressure [7]):

Rule 3: When a communication's estart changes, if there is any consumer **C** of the value communicated in a different VC than the instruction producer **P** with an lstart lower than the new estart of the communication, then: **Fuse VCs of instructions P and C**.

Rule 4: When the lstart of an instruction **I** changes, if **I** consumes a value generated by **P** in a different VC with a communication,**C**, already assigned to that value, and **C** has an estart higher than the new lstart of **I**, then: **Fuse VCs of P and C**.

Finally, a set of more advanced rules are defined to exploit the knowledge of situations where two pairs of produce-consumer exist and at least one of them must be communicated. To handle these cases *partially-linked communications* (PLC) are used.

Partially-linked, as opposed to *fully-linked* (FLC) ones, refers to communications that have not a producer or a consumer defined. The main goal for using PLC is to advance communications that will be required in the future. This knowledge helps to prevent resource conflicts with current and future communications, as well as provides more accurate information on instruction bounds. Moreover, this PLC information allows the DP to get even more benefit (further conclusions) from other rules such as the scheduling rules devoted to the resource usage study.

Some rules are used to find cases to apply PLC. In general, PLC is used when a pair of register-edges of the DG has their producers/consumers in the same cluster whereas the consumers/producers belong to a pair of incompatible clusters. Three different cases of PLC are defined: P-PLC when the producer remains to be defined, C-PLC when a consumer has not been defined, and PC-PLC for situations where neither the producer nor the consumer has been defined. An example of these rules is the following one that detects a case for P-PLC:

Rule 5: When a pair of VCs becomes incompatible, if there is a pair of instructions, each belonging to one of these VCs, with a common successor **C**, then: **a P-PLC instruction is created indicating that at least one of the values produced in the incompatible VCs will be communicated to C**.

Note that PLCs are used to describe situations where at least one of two alternative producer-consumer pairs must be assigned to a communication. At any point in the scheduling process any PLC can become a FLC.

Rule 6: When the VCs of a pair producer-consumer are fused, if this pair is involved in a PLC, then: **the PLC becomes a FLC by assigning the communication to the alternative producer-consumer pair involved in the PLC.**

Rule 7: When the VCs of a pair producer-consumer become incompatible, if this pair of instructions is involved in a PLC, then: **the PLC becomes FLC by assigning the communication to that pair**.

Finally, rules have been included to guarantee that no unnecessary partial or full communication is created. In particular, these rules handle the opportunities for communication reuse and guarantee that the DP only returns mandatory changes to the scheduling state. These rules are applied when either an opportunity for creating a new PLC is found, or a PLC becomes FLC.

An important issue of PLC is the bounds for the communications. In order to be useful for the producer-consumer pairs involved in a PLC, the communication must have enough slack to be accommodated in any alternative. Then, scheduling rules that find resource conflicts are used to handle these possibly high slacks.

4. The Proposed Algorithm

In this section, our approach to instruction scheduling and cluster assignment on superblocks for clustered architectures is presented. The novel technique makes use of the: (1) Scheduling Graph (SG), (2) Virtual Cluster Graph (VCG), and (3) Deduction Process (DP) described in Section 3.

4.1. Overview

In Figure 6, a high-level view of the proposed algorithm is shown. Initially, the SG and the VCG are generated as described in Section 3. Once both graphs have been obtained, the computation of the minAWCT is performed (see Section 4.2). Then, given a value for the AWCT, the scheduling state is initialized as described in Section 4.3. After initializing the scheduling state the process, described in Section 4.4, to find a valid schedule for the AWCT value starts.

When a final schedule is found, its correctness is checked as described in Section 4.5. If the schedule is not valid, the AWCT is increased and the process restarts with the new value. Otherwise, we are done.

4.2. AWCT

The proposed technique iteratively enumerates different values of the AWCT. Starting from the minAWCT, the AWCT is progressively increased. The value by which the AWCT is increased is the exit probability of the branch that has the lowest probability among those that its estart can be increased without requiring other branches to increase its estart.

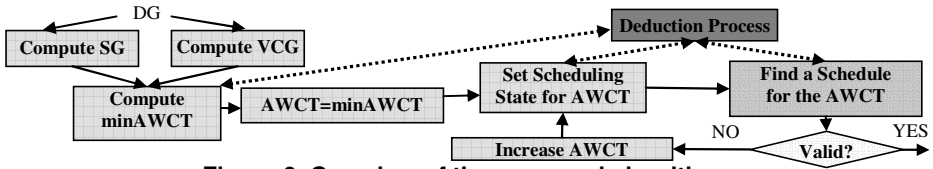


Figure 6. Overview of the proposed algorithm

The minAWCT is computed through an enhancement of the traditional computation described in Section 2. This tighter bound is computed by detecting and iteratively increasing non possible minimum distances between any branch pairs in the superblock. These distances are checked using the DP and its property of only returning mandatory constraints.

4.3. Scheduling State for the AWCT

When a new value for the AWCT is set, the scheduling state is computed accordingly. In particular, an AWCT value establishes an estart/lstart for the exit instructions, which in turn induces an estart/lstart for all instructions in the superblock.

A scheduling state is defined by: (1) the estart/lstart of each instruction, (2) a list of chosen combinations, (3) a list of discarded combinations, (4) a list of non-treated combinations, (5) a set of connected components, and, (6) the Virtual Cluster Graph.

In order to generate the initial scheduling state for the current AWCT the following actions are taken:

- Initially, the length between the entry and each exit of the superblock is set to infinite.
- All combinations are added to the list of non-treated, while the chosen and discarded lists are set to empty.
- For each instruction, a connected component and a virtual cluster is created.
- The bounds induced by the current AWCT are submitted to the DP. Since the length from the entry to each exit is initially set to infinite, the DP is able to find the consequences of these bounds and add them to the initial scheduling state.
- Finally, all combinations that result in a resource conflict are discarded using the DP.

Since any call to the DP returns only mandatory constraints, by using it to study the resource conflicts and the bounds due to the AWCT a more constrained initial scheduling state is obtained (i.e., fewer combinations can be chosen, instructions' slack decrease and there are fewer virtual clusters).

4.4. Finding a Schedule for an AWCT

The process of finding a valid schedule is divided in six stages, as shown in Figure 7. Four out of the six stages are making scheduling decisions (in light grey). The remaining ones target cluster assignment.

Starting from the scheduling state computed as described in Section 4.3, in stage 1 the technique makes a decision, choose or discard, over all combinations among instructions belonging to the original DG, i.e., instructions other than communications.

Once all combinations have been chosen or discarded, some degrees of freedom may still remain for the scheduling of some instructions. In this case, some decisions must be made regarding the cycle where each instruction is finally scheduled. In stage 2, the process deals with the bounds of these instructions, that is the non-communication instructions that have some slack.

When all non-communication instructions have been fixed to a particular cycle, the cluster assignment stages start. Initially, during the third stage decisions are made in order to eliminate outedges. An outedge refers to a value produced in one cluster and consumed in another. When no outedges remains to be treated, the VCs are finally mapped to PCs (in stage 4).

It is often the case that communications introduced at any point between stages 1 and 4 are handled, i.e., assigned to a particular cycle, during these stages. However, some of them may still have some degree of freedom. In the last two stages, communications introduced at any point in time but not treated are considered. In particular, during stage 5 decisions are made for combinations involving communications, and in stage 6 the communications with remaining slack are finally scheduled to a particular cycle.

Handling communications in earlier stages may constrain the scheduling of original instructions, and latter communications still to be inserted. On the other hand, the result of delaying this treatment to the end, as proposed, may also have negative effects coming from other instructions constraining the schedule of communications. However, we found that this latter effect is generally better treated by the DP.

At each of the stages, the iterative process shown in Figure 8 is applied. At each iteration of the process, a set of candidates is selected, studied with the DP and the best one is heuristically chosen. A candidate may be a combination (stages 1 and 5), an instruction and a cycle to schedule it (stages 2 and 6), or a set of VCs (stages 3 and 4).

The iterative process finalizes when no candidate of the type specified by the stage remains to be treated. The iterative process also finishes when a situation is found that, no matter what we do that no schedule is feasible. This occurs when a candidate can neither be chosen nor discarded. In this case, the AWCT is increased and the schedule process must restart.

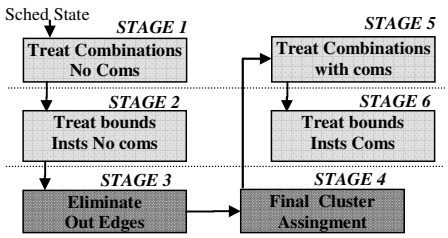


Figure 7. Stages when looking for an schedule

In the following sections the steps of the iterative process are further described.

4.4.1. Selection of candidates

At any of the six stages of the scheduling process, the most constraining candidates at that time are chosen for deep study through the DP. Depending on the stage of the algorithm, the heuristics used to select the candidates varies. In particular, three different methods have been used to select candidates: one for the instruction scheduling stages, i.e., stages 1, 2, 5 and 6; another for the elimination of outedges, i.e., stage 3; and finally, a different one for the final cluster assignment, i.e., stage 4. These methods are described next.

4.4.1.1. Instruction Scheduling Stages

The information used to identify the most constraining candidates for the scheduling stages is the slack of instructions. In particular, in case of stages 2 and 6, where the candidates are instructions, the ones selected are those that have the lowest slack.

When looking at the combination candidates, i.e., stages 1 and 5, we use the slack of the instructions involved in a combination. From the bounds of each instruction it is possible to compute the cycles where the combination may occur. The number of these cycles defines the *slack of the combination*. Combinations with the lowest slack are selected the first.

4.4.1.2. Eliminating Outedges Stage

When dealing with outedges, the proposed scheme selects pairs of VCs to be fused or to be marked as incompatible. In order to have a global view of all VCs, this stage considers a set of VC pairs at each step. To select the most appropriate set of VCs we make use of a *maximum weight matching* algorithm [1].

A matching is a set of edges selected from an undirected graph such that none of the edges in this set is adjacent. In this case, the matching algorithm is applied over the *matching graph* (MG). The MG is an undirected graph derived from the VCG. It contains a set of nodes that matches the set of VCs, and an edge for every pair of VCs with outedges among them. The VC candidates are the pairs linked by the edges belonging to the matching.

In order to have a more global view of the impact of any assignment, it is desirable to perform fusion of VCs simultaneously on the whole graph. Therefore, it is convenient that the matching contains as many edges as possible to fuse all the parts of the graph at the same time. However, there may be pairs of VCs that are better candidates than others to collapse. For this reason, each edge is given a weight and a maximum weight matching is computed. A maximum weight matching is a matching such that the sum of the weights of the edges belonging to the match is the highest possible.

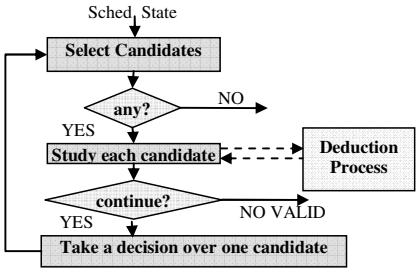


Figure 8. Process for stage

For each edge $e=(u, v)$ in the MG, its weight is computed by taking into account the number of outedges that cross each pair of virtual clusters: $Weight(e)=OutEdges(u,v)+OutEdges(v,u)$ where $OutEdges(x,y)$ is the number of outedges from x to y.

4.4.1.3. Final Mapping Stage

Once all outedges have been eliminated, the number of VCs may still exceed the number of physical ones. When this happens, additional fusion of compatible VCs is required. For this purpose, a graph coloring algorithm similar to the one applied in [4] is used. This scheme establishes an order among VCs, based on their incompatibilities. In particular, VCs are sorted based on its degree in the VCG, from the highest to the lowest.

Once the ordering among VCs is available, a VC is mapped into a physical cluster (PC) one at each step of the stage. In particular, the first VC is directly assigned to the first PC. Then, the second VC is tried to be mapped onto already used PCs, and so on. When a VC cannot be mapped onto any of the already used PCs, it is assigned to an empty PC. The candidates at this step are selected through the graph coloring algorithm.

This iterative process based on the graph coloring finishes when all VCs are assigned to a PC, or the number of PCs required exceeds the real ones, that is the process fails to find a final mapping).

4.4.2. Study of Candidates

Given a set of candidates, the DP is used to study them. The objective of this study is to produce a more accurate future scheduling state while avoiding wrong decisions.

The DP extracts mandatory consequences of applying a decision to the current state. A decision is defined by a candidate and an action to be taken with it, i.e., choose or discard: choosing a pair of VCs means to fuse them, while discarding means to set them incompatible;

choosing an instruction in a cycle means to schedule it in this cycle, while discarding an instruction in a cycle refers to not allow its schedule in this cycle.

For each candidate, the DP is called to deduce the consequences of choosing it. When according to the DP all selected candidates can be chosen, i.e., no one of them results in a contradiction, the future scheduling states obtained by DP for each candidate are compared with the heuristic criteria described in Section 4.4.3, and the best is selected. Otherwise, if any candidate results in a contradiction, it is discarded.

Note that candidates are single elements except for the stage of eliminating outedges where a candidate is a set of VC pairs. In that case, if the DP ends up with a contradiction when trying to choose the candidate, i.e., fuse all pairs, the proposed scheme does not try to discard the candidate.

In general, it is difficult to extract benefit from the knowledge obtained from discarding a set of VCs to fuse. The fact that a set of fusions is not possible does not necessary mean that every individual fusion is wrong. Only a subset of the fusions may be responsible for the contradiction. Thus, it is not mandatory to mark incompatible every fusion in the set. On the other hand, modifying the DP in a way that can benefit from this knowledge is difficult. For these reasons if a contradiction arises in the fourth stage, an alternative candidate is considered. In particular, a single pair of clusters is selected as a candidate which is the one linked by the highest weighted edge in the MG. This edge will be referred to as *E_highest_weight*.

After studying E_highest_weight edge with the usual process, i.e., first try to choose it and if failed discard it, the matching scheme resumes.

The E_highest_weight edge usually appears in the recently failed matching. Moreover, the responsibility for the contradiction of the matching is often due to this edge, since it is the most constraining one. When this occurs, treating the edge individually allows us to rule it out if it prevents the matching scheme from proceeding.

Finally, even when E_highest_weight edge is not responsible for the contradiction or it does not belong to the matching, another contradiction may arise as well when applying the DP to it.

4.4.3. Comparing Candidates

After studying the candidates through the DP, when no contradiction is found the scheduling states are compared and the best one is selected. In order to compare two future scheduling states the following heuristics are used: (1) choose the one that minimizes communications; (2) choose the one that result in more compact code; and, (3) choose the one that minimizes the ratio between outedges and virtual clusters. This is based on the observation that it is usually better to have more VCs and fewer outedges.

Note that the first two criteria are widely used in techniques for cluster scheduling. The key feature of our proposal is not a set of new heuristics, but a completely novel approach that reduces the probability of making inappropriate decisions due to the use of heuristics. This is achieved by the increased knowledge provided by the DP that feeds these heuristics.

4.5. Scheduling Correctness

A valid schedule must accomplish the following conditions: (1) all VCs have been mapped to PCs, (2) all combinations have been selected or discarded, (3) all instructions have been scheduled to a cycle, and (4) there is no pair of overlapping connected components. When a schedule does not meet all these conditions, it must be discarded, the AWCT increased and the process restarted.

5. Example of the Proposed Technique

This section shows how the DG in Figure 1 would be scheduled by the proposed technique. For this example, we use a 2-cluster machine, each cluster being able to issue: one 2-cycle latency instruction *I*, and one 3-cycle latency instruction *B* per cycle. A single 1-cycle latency bus is available to communicate values.

The technique starts by computing the minAWCT according to definition given in Section 2, which includes the schedule of B_0 in cycle 4 and B_1 in cycle 6. Then, the enhancement of this definition described in Section 4.2 finds that B_1 cannot be scheduled in cycle 6, otherwise, I_1 and I_2 must be in the same cycle, as shown in Figure 9.a. Then, one must be assigned to the first cluster and the other to the second one. However, this assignment requires a communication (PLC) that must handle the edge in the DG between I_0 and I_1, or between I_0 and I_2. Therefore, I_1 or I_2 cannot be scheduled in cycle 2 due to 1-cycle communication, and a contradiction is found. Hence, by calling the deduction process (DP) it is found that B_1 cannot be in cycle 6.

On the other hand, no constraint is found scheduling B_0 in cycle 4. Hence, the computed minAWCT is 9.1, implying that the estart of B_0 is 4 and the estart of B_1 is 7. This value is set to the AWCT, and the process of finding a valid schedule starts .

When initializing the scheduling state for an AWCT value, each instruction starts with its own VC and its own connected component, and the lengths are set to infinite. Then, the bounds for B_0 and B_1 are applied through the DP. The outcome of this call is shown in Figure 9.c, being Figure 9.b the middle point in the DP invocation where the bounds of B_0 and B_1 have been simply propagated. At this point, I_0, I_3 and B_0 belong to the same VC because no possible communication can

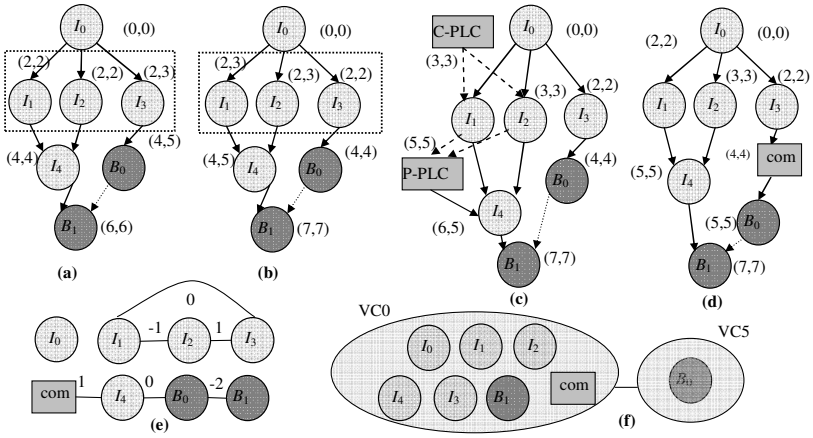


Figure 9. Example of the proposed technique

be scheduled between them. Next, the DP finds that since I_3 must be in cycle 2, I_1 and I_2 must be either scheduled: (a) in cycle 3 at the same cluster, due to the resources available; or (b) in cycle 3 at the other cluster, due to the 1-cycle needed to communicate the value produced by I_0. Hence, as shown in Figure 9.d, the DP concludes that a C-PLC must be scheduled in cycle 1, and that I_1 and I_2 must be scheduled in cycle 3. In turn I_4 must be scheduled at least in cycle 5. However, the DP also finds that cycle 5 is not a valid cycle for I_4, because of the need for a P-PLC communication relating I_1 and I_2 as possible producers, and I_1 as the consumer. This communication must be scheduled in cycle 5, and then the estart of I_4 is set to 6. This gives a contradiction with the lstart of 5. Thus, this AWCT must be discarded.

Next, the AWCT is set to 9.4 scheduling B_0 in cycle 5 and B_1 in cycle 7. In this case, the proposed technique finds a valid schedule, as shown in Figure 9.d. In order to achieve this valid schedule, the DP is used again in the initialization of the scheduling state. The outcome of this first call to DP is similar to the one shown in Figure 9.b, except that B_0 is scheduled in cycle 5 instead of cycle 4, and in turn the lstart of I_0 is increased to 1 and the lstart of I_3 is increased to 3. Then, the process of finding a valid schedule starts by considering the combinations among instructions according to stage 1 in Figure 7. In this case, two alternative candidates are considered as the most constraining ones: combinations 0 and 1 between I_4 and B_0. When studying both alternatives, the DP finds that combination 1 implies scheduling I_4 in cycle 4, and the scheduling of I_1 and I_2 both in cycle 2, without enough freedom to schedule the appropriate communications to allow them to be executed on a different cluster. Hence, combination 1 is discarded, which in this case is equivalent to chosing combination 0. Note that no decision at this point has been made heuristically, everything is mandatory.

Finally, combinations selected in stage 0 to be considered are the ones that involve I_1, I_2 and I_3. Among them, combination 0 between I_1 and I_2 must be discarded. Other alternatives are possible and the best one is chosen by the heuristics. In this case the only freedom choice whether I_1 and I_2 will be scheduled in cycles 2 or 3. In Figure 9.d the valid schedule which results from choosing the combination 0 between I_1 and I_3 is shown. All chosen combinations of this schedule at the end of stage 1 are shown in Figure 9.e (edges are labeled with the id of the combination). The VCG configuration at this point is also shown in Figure 9.f. In this case, no candidates to be handled are found for the rest of the stages other than stage 4, where the VCs shown in Figure 9.f, are finally mapped into physical clusters (e.g., VC0 into PC0 and VC7 into PC1).

6. Evaluation

6.1. Experimental Framework

The proposed algorithm has been implemented in the IMPACT compiler [5] using LEDA [22], and evaluated with more than 60,000 superblocks obtained from 7 SpecInt95 and 7 MediaBench applications. These blocks represent the whole code of each application.

Three clustered VLIW designs have been studied. For all them, a cluster has a functional unit of each type (int, fp, mem and branch) and the clusters are connected through a single bus. The first architecture is an 8-issue width machine that consists of two clusters connected by a 1-cycle latency bus. The second and the third architectures have four clusters and 16-issue width. In the case of the second architecture the latency to communicate one value from one cluster to another is 1 cycle, whereas for the third such latency is 2.

The proposed technique is compared against CARS[18], which performs instruction scheduling and cluster assignment in a single-phase based on a list scheduling scheme. For both approaches, the control flow graph of each function is traversed in a top-down fashion. For each superblock visited the DG is built and the scheduling technique is applied.

In order to perform a fair comparison between the proposed technique and CARS, values live on the entry point of each superblock and values live on each exit are randomly distributed to clusters and both use the same assignment for these values.

Experiments carried on a 1.2 GHz UltraSparc-IIIi machine show that CARS requires shorter compilation times, as shown in Figure 10. CARS compiles between 92% (for 4-clusters) and 95% (for 2-custers) of the superblocks within 1 second, and less than 1% requires more than 1 minute to generate code in our framework. The proposed technique significantly outperforms CARS in performance, as shown in the next section, although it requires additional compilation time. In particular, the proposed technique compiles between 70% (4-clusters) and 72.5% (2-clusters) of the superblocks within 1 second and less than 10% of the blocks requires more than 1 minute. Only in 1% of the blocks our approach spends more than 4 minute compiling them. This last group of blocks represents a small fraction of the whole execution time (less than 1%). Hence its contribution to the number of dynamic cycles is low enough that the use of a simpler list scheduling technique would not penalize performance. For this reason, CARS is used in for those superblocks in which our approach exceeds a threshold. Performance results considering thresholds of 1 and 4 minutes are shown in the following section.

6.2. Performance Results

In this section, we consider the total number of cycles as the main performance metric (computed as described in Section 2). The frequency of the superblock exits has been obtained through profiling. In order to obtain the profiling data, programs were run until completion using the ref input data set.

In Figure 11, the speed-up of the proposed scheme against CARS is presented for all three configurations studied, using the same input for profiling and the actual execution, and for two different thresholds: 1 minute and 4 minutes. The main conclusion that we can draw from this figure is that the scheme presented in this work produces significant gains for all configurations and for all programs with respect to CARS. On average, the schedules produced by the proposed technique when using a threshold of 4 minutes, improve the number of cycles from 2.5% to up to 9.5% over CARS. The smallest average speed-up (1.8%) is achieved for the 2-cluster architecture for SpecInt applications. This is due to the fact that this architecture offers the least opportunities for exploiting parallelism. The schedule is so constrained that there are not too many alternatives valid for the code generation. Thus, a list scheduling such as CARS is able to achieve a performance close to

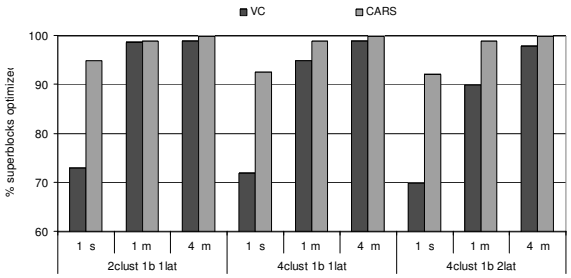


Figure 10.Compilation time comparison

the proposed technique. Moreover, we have observed that the difference between minAWCT and the AWCT that we are close to the optimal.

On the other hand, the results obtained for 4-cluster architectures, using a 4 minute timeout, show significant benefits from the proposed technique. These benefits come from the ability to exploit the additional issue width provided by four clusters while performing cluster assignment in a way that the workload is balanced among the clusters and the number of communications is minimized.

The speed-ups for a 4-cluster architecture with 2-cycle latency are remarkable. An important issue in this architecture is the treatment of the communications. The bus is not a pipelined resource, thus the complexity of the schedule increases. In this case, the speed-ups of the proposed technique are even higher than on the other configurations due to the rules in the deduction process that treat resources and PLCs. These rules allow for a better treatment of the interaction between resources and dependences, which is hard in this configuration due to the increased difficulty of hiding the latency of the communications.

If we look at the results with a timeout of 4 in more detail, we can see that for some programs (e.g., 099.go or 129.compress for the 4-cluster architecture with 2-cycle bus latency), the speed-up is close to 15%. In addition, for programs such as 130.li, 134.perl, epicdec, epicenc, or mpeg2enc, the proposed technique achieves speed-ups over 10%. The mentioned applications are clear examples of cases where the proposed technique can exploit the available issue-width while performing an appropriate cluster assignment. Moreover, when comparing the results obtained for SpecInt and MediaBench, we can see similar trends.

On the other hand, when setting the threshold to 1 mintue instead of 4, the proposed technique achieves almost the same on a 2-cluster machine, except for 147.vortex which have several complex superblocks that have some importance on the execution time. For the 4-cluster configurations results shown slightly less performance improvement compared to the case of a 4 minute's timeout. This is more noticeable for the 2-cycle bus configuration, where some media applications

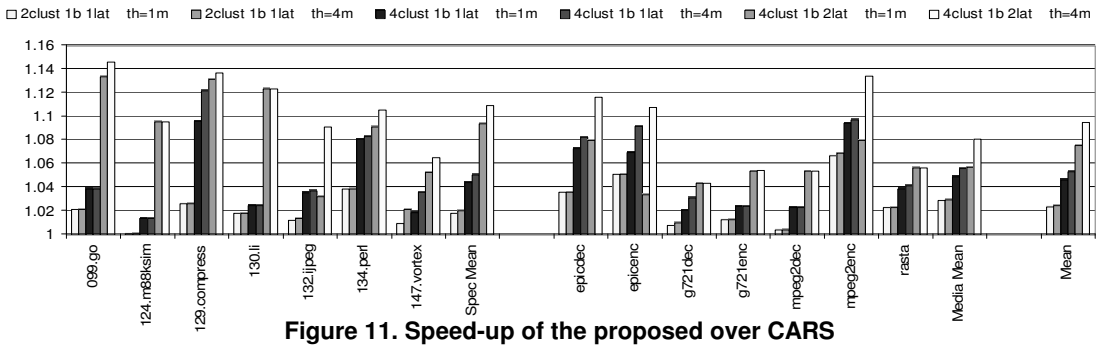


Figure 11. Speed-up of the proposed over CARS

(132.ijpeg, epicdec, epicenc and mpeg2dec) that, as 147.vortex for the 2-cluster machine, have some superblocks that cannot be scheduled within the first minute. These blocks are somewhat complex and they have noticeable contribution to the whole execution time of the application. Nevertheless, the average difference between threshold 1 and threshold 4 remains below 2% in case of the 1-cycle bus latency configuration, and below 3% for the 2-cycle one.

Finally, in Figure 12 the speedup of the proposed technique when different inputs to profile and execute are used. For this study 099.go, 132.ijpeg and 134.perl are used with 1 minute threshold. Similar trends to those find when using the same input can be observed, except for 134.perl running on a 4-cluster machine with 2-cycle bus latency. The benefits for this configuration and benchmark have been decreased due to the variations find when changing the input sets. Nevertheless, the proposed technique still outperforms CARS by 6%.

7. Related Work

Several techniques have been proposed in the literature to cluster scheduling for acyclic codes. Former proposals were based on a two-phase approach [10][3][17][9][6][20]. First, instructions are assigned to clusters, and then they are scheduled by strictly following the computed partition. These approaches use heuristics to estimate the impact on performance of clustering assignments. However, they do not consider at all the effects of the scheduling constraints imposed by the cluster decisions itself when performing further cluster assignment. Our approach deals with the interaction between both tasks through a single-phase scheme where scheduling and clustering decisions are made in a separate stage. However, the consequences of scheduling decisions (taken first), obtained through the deduction process, generates a partial assignment which at the same time constraint further scheduling decisions.

Later proposals to acyclic code regions show that performing assignment and scheduling in a single step can be more effective, since the interaction between these tasks can be taken into account [24][18][21][19]. The main drawback of these integrated schemes is that the assignment of each individual instruction is decided based only on information of the already scheduled ones, which restricts assignment to use a partial or local view of the DG. Our proposal deals with this limitation by: (1) extracting the consequences of each decision made; and, (2) postponing assignment decisions, and hence performing these assignments with accurate scheduling information.

On the other hand, Terechko et al. [27] studied the effects of global cluster assignment with a conventional list scheduling scheme. Instead, the technique proposed in this paper focus on the cluster scheduling inside a region. Our future work includes extending the proposed scheme to consider global value assignment.

Finally, several proposals to cluster scheduling based on modulo scheduling have been proposed [23][12][26] [7] [29][1][2]. These proposals focus on loops, whereas our technique targets general code regions. Hence, the goals and the heuristics used are different.

8. Conclusions

In this paper we have presented a novel approach to superblock scheduling for clustered microarchitectures. The proposed technique produces high performance codes where parallelism is exploited, workload among clusters is balanced and the number of communications is minimized. This is achieved by: (1) delaying the final cluster assignment until the end of the scheme, where complete information of the final schedule is available; while, (2) using partial cluster assignment to assess the scheduling process.

The proposed technique is a single-step approach that makes use of scheduling graphs and virtual clusters. These novel representations are managed through a powerful deduction process. The use of the deduction process allows for a better understanding of the consequences of each change on the scheduling state, which helps to prevent from making non-appropriate or poor decisions.

The described technique is shown to outperform a state-of-the-art scheduler for all programs and for all configurations evaluated. Some of the speed-ups achieved are close to 15%, while 10% speed-up is reached for many programs. Average speed-ups range from 2.5% (2-clusters) to up to 9.5% (4-clusters).

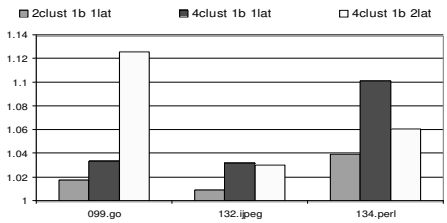


Figure 12. Results using different inputs

9. Acknowledgements

This work was partially supported by the Spanish Ministry of Education and Science under contract TIN2004-03072 and Feder funds and Intel. The authors would like to thank Cliff Young and the anonymous reviewers for their helpful comments and suggestions. We also thank Enric Gibert for his work with IMPACT; as well as Kemal Ebcioglu and Krishnan Kailas for their help to implement CARS.

10. References

[1] A. Aletà, J.M. Codina, J. Sánchez, A. González, and D. Kaeli, "Exploiting Pseudo-schedules to Guide Data Dependence Graph Partitioning", in *Proc. of the 2002 Int. Conf. on Parallel Architectures and Compiler Techniques*.

[2] A. Aletà, J.M. Codina, A. González, and D. Kaeli, "Instruction Replication for Clustered Microarchitectures", in *Proc. of 36th Int. Symp. on Microarchitecture,* 2003.

[3] A. Capitanio, D. Dytt, and A. Nicolau, "Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs", in *Proc. of 25th. Int. Symp. on Microarchitecture*, 1992.

[4] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein, "Register Allocation Via Coloring", *Computer Language*s, 1981.

[5] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Water, and W.W. Hwu, "IMPACT: An Architectural Framework for Multiple- Instruction-Issue Processors", in *Proc. of the 18th Int. Symp. on Computer Architecture*, 1991.

[6] M. Chu, K. Fan, and S. Mahlke, "Region-based Hierarchical Operation Partitioning for Multicluster Processors", in *Proc Conf. on Programming Languages and Implementation Design*, 2003.

[7] J.M. Codina, J. Sánchez, and A. González, "A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors", in *Proc. of the 2001 Int. Conf. on Parallel Architectures and Compilation Techniques*.

[8] J.M. Codina, J. Sánchez, and A. González, "Virtual Cluster Scheduling Through the Scheduling Graph", Tech. Report UPC-DAC-RR-ARCO-2005-6, Dep. of *Computer Architecture, UPC*, Barcelona.

[9] G. Desoli, "Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach", *Technical Report HPL-98-13*, HP Laboratories, 1998.

[10] R. Ellis, "Bulldog: A Compiler for VLIW Architectures", *MIT Press*, pp. 180-184, 1986.

[11] P. Faraboschi, G. Brown, J. Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing", in *Proc. of the 27th Int. Symp on Computer Architecture*, 2000.

[12] M.M. Fernandes, J. Llosa and N. Topham, "Distributed Modulo Scheduling", in *Procs. of Int. Symp. on High-Performance Computer Architecture*, Jan. 1999.

[13] J. Fridman, and Z. Greenfield, "The TigerSharc DSP Architecture", *IEEE Micro*, pp. 66-76, Jan-Feb. 2000.

[14] P.N. Glaskowsky, "MAP1000 unfolds at Equator", *Microprocessor Report*, 12(16), Dec. 1998.

[15] R. Ho, K. Mai, and M. Horowitz, "The Future of Wires", in *Proc. of IEEE*, April 2001.

[16] W. Hwu et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *Journal of Supercomp.*, v.7 n.1/2, 1993.

[17] S. Jang, S. Carr, P. Sweany, and D. Kuras, "A Code Generation Framework for VLIW Architectures with Partitioned Register Banks", in *Proc. of 3rd. Int. Conf. on Massively Parallel Computing Systems*, 1998.

[18] K. Kailas, K. Ebcioglu, and A. Agrawala, "CARS: A New Code Generation Framework for Clustered ILP Processors" in *Proc. of the 7th Int. Symp. on High-Performance Computer Architecture*, 2001.

[19] K.V. Lakshmi, D. Sreedhar, E. Raman and P. Shankar, "Integrating a New Cluster Assignment and Scheduling Algorithm into an Experimental Retaragable Code Generation Framework", in *Proc. of the Int. Conf. on High Performance Computing (HiPC)*, 2005.

[20] V.S. Lapinskii, M. F. Jacome, G.A. Veciana, "Cluster Assignment for High-Performance Embedded VLIW Processors", in *ACM Trans. on Design Automation of Electronic Systems*, Vol. 7, No. 3, July 2002.

[21] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe, "Convergent Scheduling", in *Proc. of 35th Int. Symp. on Microarchitecture*, 2002.

[22] K. Mehlhorn and S. Näher, "LEDA, a library of efficient data types and algorithms", Tech. Report TR A 04/89, *Universität des Saarlandes, Saarbrücken*, 1989.

[23] E. Nystrom and A. E. Eichenberger, "Effective Cluster Assignement for Modulo Scheduling", in *Procs. of the 31st Int. Symp. on Microarchitecture*, 1998.

[24] E. Özer, S. Banerjia, and T. Conte, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures", *In Procs. of the 31st Int. Symp. on Microarchitecture*, 1998.

[25] G.G. Pechanek, and S. Vassiliadis, "The ManArray Embedded Processor Architecture", in *Proc. of the 26th. Euromicro Conference: "Informatics: inventing the future"*, Maastricht, The Netherlands, 2000.

[26] J. Sánchez and A. González, "The Effectiveness of Loop Unrolling for Modulo Scheduling in Clustered VLIW Architectures", in *Procs. of the 29th Int. Conf. on Parallel Processing*, Aug. 2000.

[27] A. Terechko, E. Le Thenaff, and H. Corporaal, "Cluster assignment of global values for clustered VLIW processors", in *Proc. of the 2003 Int.Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*.

[28] Texas Instruments Inc. "TMS320C62x/67x CPU and Instruction Set Reference Guide", 1998.

[29] J. Zalamea, J. Llosa, E. Ayguadé and M. Valero, "Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures", in *Proc. of 34th Int. Symp. on Microarchitecture*, Dec 2001.